A Practical Query Language for Graph DBs

Renzo Angles[1,2], Pablo Barceló[3], and Gonzalo Ríos[3]

[1] Department of Computer Science, Universidad de Talca
[2] Department of Computer Science, VU University Amsterdam
[3] Department of Computer Science, Universidad de Chile

Abstract. Query languages for current graph DB systems lack clear syntax and semantics, which difficults the understanding of its expressiveness and complexity. In particular, many of them suffer from poor performance due to the inherently high complexity of the queries they can express. We propose propositional dynamic logic (PDL) as a yardstick query language for graph database engines, based on the fact that it can express many relevant properties with very low computational cost. We present an implementation of the language that shows its potential applicability for querying massive graph databases by building on existing graph database support.

1 Introduction

Some of the current systems for managing graph data implement APIs with special functions for querying graph properties (e.g., Dex and Titan). Others include graph-oriented query languages; e.g., Neo4j provides Cypher[4], based on expressions of the form *start-match-where-return*; OrientDB[5] includes a SQL-style language extended for querying graphs; InfiniteGraph[6] allows navigation through the implementation of Java classes; and RDF stores like AllegroGraph, Virtuoso and BigData support SPARQL[7], the standard query language for RDF.

But with the exception of SPARQL, none of the above languages provides a formal syntax and semantics, which difficults the accurate evaluation of their expressive power and complexity. Moreover, after some empirical experiments (not included here due to lack of space), we found that many of them suffer from poor performance. This is not due to the implementation, but to the inherently high computational complexity of the queries they allow to express.

We propose a *navigational* language – originally designed for program verification – as a yardstick for graph database engines. The language is *propositional dynamic logic* [5], PDL, that extends several important graph database languages [3]. The reason is that the language combines good properties of evaluation and expressiveness: It can be evaluated in polynomial time, and even in linear time for an important fragment of graph queries. In addition, it allows to express relevant properties of graph databases, as we will see soon.

[4] http://docs.neo4j.org/chunked/milestone/cypher-query-lang.html
[5] http://www.orientdb.org
[6] http://http://objectivity.com
[7] http://www.w3.org/TR/rdf-sparql-query/

We also present an implementation of the language that works reasonably well, but decreases its performance for large graph databases in comparison with other engines (more specifically, DEX) that have much better support. The conclusion we draw is that implementation of PDL for querying massive graph databases seems promising, but it can only be achieved by building on existing graph database engines.

2 The Query Language

We work with a simple graph data model that lies at the core of most graph data models studied in the literature [2]. In essence, our graph databases are edge-labeled directed graphs, in which each element (node) is attached a single attribute with its corresponding value. We formalize this below.

Let $\mathcal{V}$ be a countably infinite set of node ids and Str the set of strings over some alphabet. Let Σ be a finite alphabet. A *graph database* G over Σ is a tuple $(V, E, @)$ such that: (1) V is a finite set of node ids (i.e., elements in $\mathcal{V}$), called the *nodes* of G, (2) $E \subseteq V \times \Sigma \times V$ is the set of labeled *edges* of G, and (3) $@Att : V \to \mathsf{Str}$ is the *attribute-value assignment* of G. The intuitive interpretation behind an edge $(u, c, v) \in E$, for $u, v \in V$ and $c \in \Sigma$, is that there exists a c-labeled edge from u to v in G. Also, $@Att(v) = k$, for a node $v \in V$ and a string $k \in \mathsf{Str}$, means that the single attribute $@Att$ of node v in G is assigned value k.

The query language for graph databases we propose is the extension of *propositional dynamic logic* [5], PDL, with the *inverse* operator [8]. This extension allows to increase the expressive power of the language without computational cost. The syntax of the language is as follows. Recall that $\mathcal{V}$ is a countably infinite set of node ids over which nodes of graph databases are taken. Let Σ be a finite alphabet. The language PDL with converse (PDL^-) over Σ is defined by the following grammar, in which α denotes *programs* and ϕ denotes *formulas*:

$$\alpha := \epsilon \mid c\,(c \in \Sigma) \mid c^-\,(c \in \Sigma) \mid \alpha \cup \alpha \mid \alpha \cdot \alpha \mid \alpha^* \mid \phi?$$
$$\phi := \top \mid [\downarrow v]\,(v \in \mathcal{V}) \mid [@Att = k]\,(k \in \mathsf{Str}) \mid \phi \wedge \phi \mid \neg\phi \mid \langle\alpha\rangle\phi.$$

We now formalize the semantics. Let $G = (V, E)$ be a graph database over Σ (that is, $V \subseteq \mathcal{V}$). Each program α defines a binary relation $[\![\alpha]\!]_G$ on V. Analogously, each formula ϕ defines over G a subset $[\![\phi]\!]_G$ of V. The definitions of $[\![\alpha]\!]_G$ and $[\![\phi]\!]_G$ are mutually inductive. We start with the case of programs. We assume that c belongs to Σ, that α, α_1 and α_2 are programs, and that ϕ is a formula: (1) Basis cases: (a) $[\![\epsilon]\!]_G = \{(v, v) \mid v \in V\}$, (b) $[\![c]\!]_G = \{(u, v) \mid (u, c, v) \in E\}$, and (c) $[\![c^-]\!]_G = \{(u, v) \mid (v, c, u) \in E\}$. (2) Inductive cases: (a) $[\![\alpha_1 \cup \alpha_2]\!]_G = [\![\alpha_1]\!]_G \cup [\![\alpha_2]\!]_G$, (b) $[\![\alpha_1 \cdot \alpha_2]\!]_G = [\![\alpha_1]\!]_G \circ [\![\alpha_2]\!]_G$, (c) $[\![\alpha^*]\!]_G = [\![\epsilon]\!]_G \cup [\![\alpha]\!]_G \cup ([\![\alpha]\!]_G \circ [\![\alpha]\!]_G) \cup \cdots$, and (d) $[\![\phi?]\!]_G = \{(u, u) \mid u \in [\![\phi]\!]_G\}$. Here, $\circ$ denotes the usual composition of binary relations. That is, $[\![\alpha_1]\!]_G \circ [\![\alpha_2]\!]_G$ is the set of pairs (u, v) such that $(u, w) \in [\![\alpha_1]\!]_G$ and $(w, v) \in [\![\alpha_2]\!]_G$, for some $w \in V$.

Let us provide some intuition for the semantics of programs: ϵ defines the identity on $V \times V$, the pairs of nodes linked by a c-labeled edge are defined by the expression c, and c^- defines the inverse of c. Definable binary relations are closed under union,

composition and transitive-reflexive closure, which are represented by operators $\cup$, $\cdot$ and $(\,)^*$, respectively. Finally, $\phi?$ defines the set of pairs (u, u) such that u satisfies the formula ϕ.

The semantics of formulas is defined as follows. We assume that $v \in \mathcal{V}$, $k \in \mathsf{Str}$, α is a program, and ϕ, ϕ_1 and ϕ_2 are formulas: (1) Basis cases: (a) $[\![\top]\!]_G = V$, (b) $[\![[\downarrow v]]\!]_G = \{v\}$, if $v \in V$, and $[\![[\downarrow v]]\!]_G = \emptyset$, otherwise, and (c) $[\![[@Att = k]]\!]_G = \{v \in V \mid @Att(v) = k\}$. (2) Inductive cases: (a) $[\![\phi_1 \wedge \phi_2]\!]_G = [\![\phi_1]\!]_G \cap [\![\phi_2]\!]_G$, (b) $[\![\neg\phi]\!]_G = V \setminus [\![\phi]\!]_G$, and (c) $[\![\langle\alpha\rangle\phi]\!]_G = \{u \mid (u, v) \in [\![\alpha]\!]_G, \text{ for some } v \in [\![\phi]\!]_G\}$.

The intuition behind the semantics of formulas is as follows: $\top$ defines the whole set of vertices, $[\downarrow v]$ is true only at the node id v, and $[@Att = k]$ defines the set of nodes whose attribute value is k. Definable sets are closed under Boolean operations, represented by operators $\wedge$ and $\neg$. Finally, $\langle\alpha\rangle\phi$ defines the set of nodes u from which a node v that satisfies ϕ can be "reached" using program α.

Example 1. Let us consider a toy example of a social network over alphabet $\Sigma = \{\texttt{friend}\}$, where nodes are persons and attributes denote their names. The query that retrieves all friends of person p is definable in our language by the following expression: $\langle\texttt{friend}\rangle[\downarrow p]$. Intuitively, this expression defines the set of persons p' in the social network that are adjacent via a `friend`-labeled edge to another person p'' (or, formally, the pair (p', p'') satisfies the program $\langle\texttt{friend}\rangle$), and the id of p'' is p (or, formally, p'' satisfies the formula $[\downarrow p]$).

Closure of formulas under Boolean combinations allows us to express important properties. For instance, the expression $\langle\texttt{friend}\rangle[\downarrow p] \wedge \langle\texttt{friend}\rangle[\downarrow p']$ defines the common friends of p and p'.

The use of regular expressions helps expressing interesting navigational properties of graph databases. For instance, the expression $\langle\texttt{friend}^*\rangle[\downarrow p]$ defines the people that is connected by a friendship sequence to person p, i.e., the people who knows someone who knows someone ... who knows p.

The language also allows to talk about the inverse of a relation, which is useful when relationships – unlike friendship in a social network – are not bidirectional. Assume, for instance, that Σ is now extended with a new symbol `parent`, that defines the set of pairs (p, p') such that p is a parent of p'. Then the expression $\langle\texttt{parent}^- \cdot \texttt{parent}\rangle[\downarrow p]$ defines the set of siblings of p.

Finally, the combination of features of the language and the use of attributes allows us to express some sophisticated queries. For instance, the expression $\langle\big(\texttt{friend} \cdot (\texttt{parent}[@ = \text{John}])?\big)^*\rangle[\downarrow p]$ defines the set of persons that are linked by a friendship sequence to p, in such a way that each person in the sequence has a son named John. □

Expressiveness and complexity In the above sections we presented examples of relevant properties of graph databases that can be expressed in PDL^-. The language also subsumes some important *navigational* query languages for graph databases that have been studied in the literature, e.g., *nested regular expressions* [3], that were originally proposed for querying Semantic Web data [7], and a tailored version of the XML query language *XPath* for querying graph data [6].

Expressions in PDL^- are *acyclic*, i.e., they cannot express interesting properties about cycles in the underlying graph database. For instance, consider again the case of

social networks over alphabet $\Sigma = \{\texttt{friend}, \texttt{parent}\}$. There is no PDL^- formula ϕ such that for each graph database G over Σ it is the case that $[\![\phi]\!]_G$ coincides with the set of persons that have two friends, one of which is a parent of the other [3]. This shortcoming of the language is at the service of efficiency: Allowing cycles in queries easily leads to intractability of evaluation [1], while we will see next that evaluation of expressions in PDL^- is tractable.

The language PDL^- has good properties in terms of *evaluation complexity*, that is, the theoretical cost of computing $[\![\alpha]\!]_G$ and $[\![\phi]\!]_G$, for a PDL^- program α and a PDL^- formula ϕ, respectively, over a graph database G. This is confirmed by the following folklore result that can be proved using standard *model checking* techniques [4]. Here, $|G|$, $|\phi|$ and $|\alpha|$ denote the size of a reasonable encoding of a graph database G, a PDL^- formula ϕ, and a PDL^- program α, respectively:

Theorem 1. *1. The cost of computing the set $[\![\phi]\!]_G$, for G a graph database and ϕ a PDL^- formula over the same alphabet Σ, is $O(|G| \cdot |\phi|)$.*
2. The cost of computing the set $[\![\alpha]\!]_G$, for G a graph database and α a PDL^- program over the same alphabet Σ, is $O(|G|^2 \cdot |\phi|)$.

In practice, specifications (formulas and programs) are much smaller than the data where they are evaluated (the graph database). Under such view, the previous result essentially tells us that PDL^- formulas can be evaluated in linear time in the size of the data, and that programs can be evaluated in quadratic time in the size of the data. The quadratic running time for evaluating programs is optimal, since in the worst case a program can define the whole set of pairs of nodes of a graph database.

3 Implementation and Evaluation

Implementation: We wanted our implementation to work on reasonably large graph databases, and, thus, we decided to concentrate on the evaluation of PDL^- formulas, as they have linear time complexity. In fact, a quadratic running time for program evaluation may be rendered as unfeasible unless deep and novel optimization techniques are used in the implementation. Our main assumption, based on the size of the graph databases we want to query, is that main memory structures used in the implementation must be of size at most $O(|V|)$ (i.e., only boolean operations on nodes may be handled there), while external memory structures are of size $O(|G|)$ (that is, the graph database is kept on disk).

In order to minimize the access to external memory, we identified a minimal set of operations that have to be handled there. These are: Given a set V_0 of nodes, compute the set V_1 of nodes that have a c-labeled edge to a node in V_0, and the set V_2 of nodes that can be reached by a c-labeled edge from a node in V_0. We used the following data structure to implement these operations: For each label c we have an array that contains an element for each node $v \in V$. Each such element is a linked list containing the c-neighbors of the node. This structure can not be explicitly implemented in external disk, but it can be emulated as follows (nodes are represented by *long* integers): (1) For each label *label* we create two files: *label.gdb* y *label.aux*. These files consist of lines of fixed size. (2) The k-th line of *label.gdb* stores data related to k-th node. (3) Each line

	G1	G2	G3
Nodes	10.000	100.000	1.000.000
Edges	43.547	550.625	6.657.553
PDL	25462 ms / 18.4MB	212422 ms / 168MB	2329189 ms / 1.7GB
Dex	571 ms / 6.9MB	5522 ms / 47.6MB	77590 ms / 490MB
Neo4j	7569 ms / 4.4MB	54360 ms / 51.7MB	875406 ms / 587MB

Table 1. Results of loading graphs of three sizes in PDL, Dex and Neo4j. For each system we show the loading time and the disk space used for data storing.

of *label.gdb* will have n consecutive *longs*, where the first and last *long* will be used as a node identifier and a pointer to the auxiliary file, respectively. The $n-2$ remaining *longs* will store the neighboring nodes of the respective node. (4) Once these $n-2$ *longs* are occupied, we create a new empty line with m longs at the end of *label.aux*, and we assign the pointer of *label.gdb* to this new line. (5) The first $m-1$ *longs* will store the neighbors of the respective node, and the last one will be used as a pointer in the same way than in the file *label.gdb*. (6) For the inverse *label*$^-$, we analogously create the files *label_gdb* y *label_aux*.

If pointers in the files were arranged so that the disk access was sequential, then the algorithm would be optimal. Unfortunately, the way the data is ordered depends on the order of insertion, which can not be known a priori. But notice that in *label.aux* pointers are always higher than the line they point to, that is, if in the k-th line pointer is p, then $p > k$. The idea of the algorithm is, thus, to keep in main memory a data structure, called *LazyP*, that stores the "pending accesses" to the lines in *label.aux*. The pseudo-algorithm is the following: (1) Read sequentially all lines in *label.gdb*, where the k-th line corresponds to the k-th node. If any of the $n-2$ nodes belongs to V_0, add the node k to the output, else if the pointer p is not 0, add the pair (p, k) to *LazyP*. (2) Iterate *LazyP* in ascending order in p, and read the p-th line in *label.aux*. If any of the $m-1$ nodes belongs to V_0, add the k-th node to the output, else if the new pointer $\bar{p}$ is not 0, add the pair $(\bar{p}, k)$ to *LazyP*.

In the first part of the algorithm the file *label.gdb* is read sequentially, and in the second part the file *label.aux* is read sequentially, which is optimal.

Evaluation results : We present an experimental evaluation of the implementation of our query language. The objective is to show the performance of PDL for loading and querying several sizes of data, and a referential comparison with two well-know graph databases, Dex and Neo4j.

All the experiments were conducted on a PC with 7 processors Intel Core i7-2600 of 3.4GHz, 15.6 GB RAM, running a Fedora Linux 64-bits. The execution of the java programs were done by using the **j**ava -jar command with parameter `-Xmx10000m` in order to set the maximum heap memory size used by Java.

We use a social network data use-case consisting of people and Webpages. A person has attributes `id` and `name`, and a Web page has attribute `id`. Two people can be related by an undirected edge `friend`, and a person can be connected with a Web page via a directed edge `like`. The data follows a power-law distribution for both relations (e.g,

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Total Time
G1											
PDL	7	126	69	4	131	146	81	382	88	87	1121
Dex	1121	6	5	1	31	6	4	78	1	2	1255
Neo4j	343	282	322	84	5630	218	197	34	332	196	7638
G2											
PDL	1421	201	260	41	3208	251	106	2182	150	109	7929
Dex	1503	25	50	1	117	30	14	162	2	2	1926
Neo4j	438	361	401	86	7537	221	225	1237	491	200	11197
G3											
PDL	7014	2166	1237	1062	4704	416	891	30828	6835	6304	61457
Dex	8233	274	6698	2825	2127	281	83	3863	25	2	24411
Neo4j	5767	1897	4411	88	9115	210	270	12319	14070	291	48438

Table 2. Results of evaluating the query mix over graphs of 10K (G1), 100K (G2) and 1M (G3) nodes. For each query we include the time (in milliseconds) of executing 100 instances of the query. The last column shows the total time of executing.

there are a small number of people having a lot of friends, and most people have a reduced number of friends). The datasets were created using the generator available at http://dcc.utalca.cl/∼rangles/research/gdg/.

The evaluation considered loading and querying tests for graphs of three sizes. Table 1 shows the number of nodes and edges for each size, the loading time and the space on disk occupied for each system after data loading. Notice that PDL presents the highest loading time, and it spends a lot of disk space for storing its data structures in comparison with Dex and Neo4j that have better support for this task.

For the query processing test, we used the following query set:

- (Q1) Get people having name N: $[@\mathrm{name} = \mathrm{N}]$.
- (Q2) Get people that likes a given Web page W: $\langle \texttt{like} \rangle [@\mathrm{id} = \mathrm{W}]$.
- (Q3) Get the Web pages that person P likes: $\langle \mathrm{likes}^{-} \rangle [@\mathrm{id} = \mathrm{P}]$.
- (Q4) Check if N is the name of person P: $[@\texttt{id} = \mathrm{P}] \wedge [@\mathrm{name} = \mathrm{N}]$.
- (Q5) Get the friends of the friends of person P: $\langle \texttt{friend} \rangle \big(\langle \texttt{friend} \rangle [@\mathrm{id} = \mathrm{P}] \big)$.
- (Q6) Get the Web pages liked by the friends of a given person P:

$$\langle \texttt{like}^{-} \rangle \big(\langle \texttt{friend} \rangle [@\mathrm{id} = \mathrm{P}] \big).$$

- (Q7) Get people that likes a Web page which a person P likes:

$$\langle \texttt{like} \rangle \big(\langle \texttt{like}^{-} \rangle [@\mathrm{id} = \mathrm{P}] \big).$$

- (Q8) Is there a "friend" connection (path) between person P1 and P2?

$$[@\mathrm{id} = \mathrm{P1}] \wedge \langle \texttt{friend}^{*} \rangle [@\mathrm{id} = \mathrm{P2}].$$

- (Q9) Get the common friends between people P1 and P2:

$$\langle \texttt{friend} \rangle [@\mathrm{id} = \mathrm{P1}] \wedge \langle \texttt{friend} \rangle [@\mathrm{id} = \mathrm{P2}].$$

– (Q10) Get the common Web pages that people P1 and P2 like:

$$\langle \texttt{like}^- \rangle [@\mathrm{id} = \mathrm{P1}] \wedge \langle \texttt{like}^- \rangle [@\mathrm{id} = \mathrm{P2}].$$

This query mix is oriented to evaluate the support of essential graph queries, that is: attribute searching (Q1 and Q4), node/edge adjacency (Q2 and Q3), fixed-length paths (Q5, Q6 and Q7), reachability (Q8) and graph pattern matching (Q9 and Q10).

Table 2 shows the results of evaluating the test queries for the graphs G1, G2 and G3. The table shows the time of evaluating each query 100 times, and the total time of the query mix. According to this table we can see that PDL^- works better than Dex and Neo4j just one time (Q1 for G1), rarely is better than Dex (Q1 for G1, G2 and G3), and several times is better than Neo4j. Note that Dex is the system with the best performance in the test.

These results show that our current implementation of PDL works well for small graphs, but its performance decreases when the data size grows. This suggests that a potential application of PDL^- for querying large graph databases seems promising, but this has to be accompanied by the support of existing graph database engines.

4 Conclusions and Future Work

In this paper we have proposed a yardstick language for querying graph databases that supports relevant graph queries and can be evaluated in linear time. In the future we plan to consider the definition of a high-level syntax for the language, in order to facilitate the construction and understanding of complex queries by the general user. Although the simplest approach would be considering an extension of the well-known SQL syntax, we hope to explore and design a more interesting syntax based on graph structures but enforcing the restrictions of PDL formulas.

Acknowledgements: Angles is funded by Fondecyt grant 11100364 and Barceló by Fondecyt grant 1130104.

References

1. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. Renzo Angles and Claudio Gutiérrez. Survey of graph database models. *ACM Computing Surveys*, 40(1), 2008.
3. Pablo Barceló, Jorge Pérez, and Juan Reutter. Relative expresiveness of nested regular expressions. In *Alberto Mendelzon Workshop, AMW*, pages 180–195, 2012.
4. Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1st edition, 1999.
5. Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979.
6. Leonid Libkin, Wim Martens, and Domagoj Vrgoc. Querying graph databases with xpath. In *International Conference on Database Theory, ICDT*, 2013.
7. Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. nSPARQL: A navigational language for RDF. *J. Web Sem.*, 8(4):255–270, 2010.
8. Moshe Y. Vardi. The taming of converse: Reasoning about two-way computations. In *Logic of Programs*, pages 413–423, 1985.